

React 事件系统

那为什么要自定义一套事件系统？

在研究一个事物之前，我首先要问为什么？了解它的动机，才有利于你对它有本质的认识。

React 自定义一套事件系统的动机有以下几个：

- 1. **抹平浏览器之间的兼容性差异。** 这是估计最原始的动机，React 根据 [W3C 规范](#)来定义这些合成事件(SyntheticEvent)，意在抹平浏览器之间的差异。

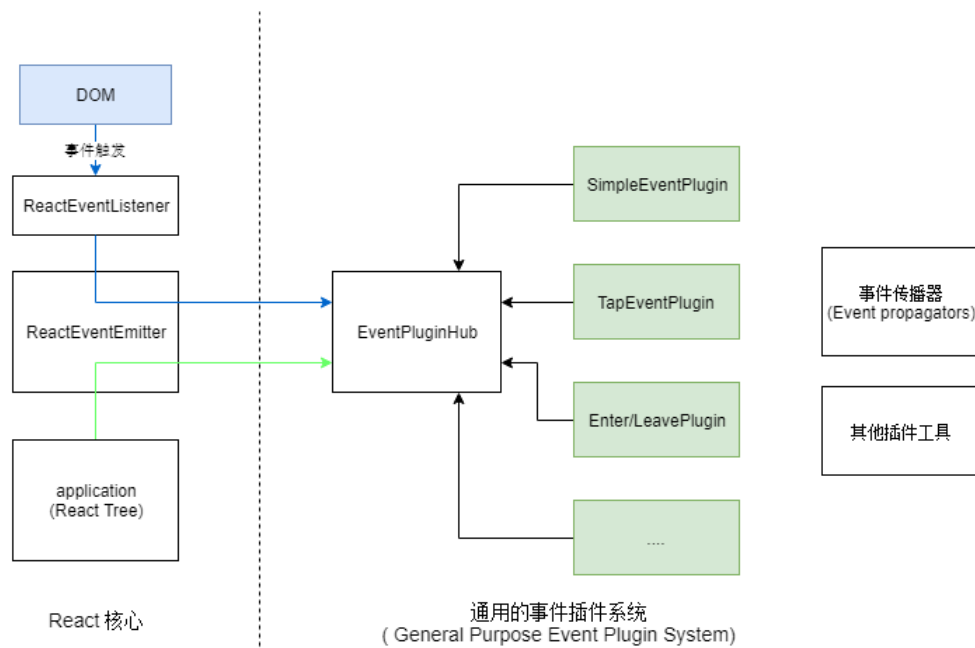
另外 React 还会试图通过其他相关事件来模拟一些低版本不兼容的事件，这才是‘合成’的本来意思吧？。

- 2. **事件‘合成’，即事件自定义。**事件合成除了处理兼容性问题，还可以用来自定义高级事件，比较典型的是 React 的 onChange 事件，它为表单元素定义了统一的值变动事件。另外第三方也可以通过 React 的事件插件机制来合成自定义事件，尽管很少人这么做。
- 3. **抽象跨平台事件机制。** 和 VirtualDOM 的意义差不多，VirtualDOM 抽象了跨平台的渲染方式，那么对应的 SyntheticEvent 目的也是想提供一个抽象的跨平台事件机制。
- 4. **React 打算做更多优化。**比如利用事件委托机制，大部分事件最终绑定到了 Document，而不是 DOM 节点本身。这样简化了 DOM 事件处理逻辑，减少了内存开销。但这也意味着，**React 需要自己模拟一套事件冒泡的机制。**
- 5. **React 打算干预事件的分发。**v16 引入 Fiber 架构，React 为了优化用户的交互体验，会干预事件的分发。不同类型的事件有不同的优先级，比如高优先级的事件可以中断渲染，让用户代码可以及时响应用户交互。

Ok，后面我们会深入了解 React 的事件实现，我会尽量不贴代码，用流程图说话。

基本概念

整体的架构



- **ReactEventListener** - 事件处理器。在这里进行事件处理器的绑定。当 DOM 触发事件时，会从这里开始调度分发到 React 组件树
- **ReactEventEmitter** - 暴露接口给 React 组件层用于添加事件订阅
- **EventPluginHub** - 如其名，这是一个‘插件插槽’，负责管理和注册各种插件。在事件分发时，调用插件来生成合成事件
- **Plugin** - React 事件系统使用了插件机制来管理不同行为的事件。这些插件会处理自己感兴趣的事件类型，并生成合成事件对象。目前 ReactDOM 有以下几种插件类型：
 - **SimpleEventPlugin** - 简单事件，处理一些比较通用的事件类型，例如 click、input、keyDown、mouseOver、mouseOut、pointerOver、pointerOut
 - **EnterLeaveEventPlugin** - mouseEnter/mouseLeave 和 pointerEnter/pointerLeave 这两类事件比较特殊，和 *over/*out 事件相比，它们不支持事件冒泡，*enter 会给所有进入的元素发送事件，行为有点类似于: hover；而*over 在进入元素后，还会冒泡通知其上级。可以通过这个[实例](#)观察 enter 和 over 的区别。

如果树层次比较深，大量的 `mouseenter` 触发可能导致性能问题。另外其不支持冒泡，无法在 `Document` 完美的监听和分发，所以 `ReactDOM` 使用 `*over/*out` 事件来模拟这些 `*enter/*leave`。

- **ChangeEventPlugin** - `change` 事件是 `React` 的一个自定义事件，旨在规范化表单元素的变动事件。

它支持这些表单元素：`input`, `textarea`, `select`

- **SelectEventPlugin** - 和 `change` 事件一样，`React` 为表单元素规范化了 `select` (选择范围变动) 事件，适用于 `input`、`textarea`、`contentEditable` 元素。
- **BeforeInputEventPlugin** - `beforeinput` 事件以及 [composition](#) 事件处理。

本文主要会关注 `SimpleEventPlugin` 的实现，有兴趣的读者可以自己阅读 `React` 的源代码。

- **EventPropagators** 按照 DOM 事件传播的两个阶段，遍历 `React` 组件树，并收集所有组件的事件处理器。
- **EventBatching** 负责批量执行事件队列和事件处理器，处理事件冒泡。
- **SyntheticEvent** 这是‘合成’事件的基类，可以对应 DOM 的 `Event` 对象。只不过 `React` 为了减低内存损耗和垃圾回收，使用一个对象池来构建和释放事件对象，也就是说 `SyntheticEvent` 不能用于异步引用，它在同步执行完事件处理器后就会被释放。

`SyntheticEvent` 也有子类，和 DOM 具体事件类型一一匹配：

- `SyntheticAnimationEvent`
- `SyntheticClipboardEvent`
- `SyntheticCompositionEvent`
- `SyntheticDragEvent`
- `SyntheticFocusEvent`
- `SyntheticInputEvent`
- `SyntheticKeyboardEvent`
- `SyntheticMouseEvent`
- `SyntheticPointerEvent`
- `SyntheticTouchEvent`
-

事件分类与优先级

SimpleEventPlugin 将事件类型划分成了三类，对应不同的优先级(优先级由低到高)：

- **DiscreteEvent** 离散事件。例如 blur、focus、click、submit、touchStart。这些事件都是离散触发的
- **UserBlockingEvent** 用户阻塞事件。例如 touchMove、mouseMove、scroll、drag、dragOver 等等。这些事件会‘阻塞’用户的交互。
- **ContinuousEvent** 可连续事件。例如 load、error、loadStart、abort、animationEnd。这个优先级最高，也就是说它们应该是立即同步执行的，这就是 Continuous 的意义，即可连续的执行，不被打断。

可能要先了解一下 React 调度(Schedule)的优先级，才能理解这三种事件类型的区别。截止到本文写作时，React 有 5 个优先级级别：

- Immediate - 这个优先级的任务会同步执行，或者说要马上执行且不能中断
- UserBlocking(250ms timeout) 这些任务一般是用户交互的结果，需要即时得到反馈。
- Normal (5s timeout) 应对哪些不需要立即感受到的任务，例如网络请求
- Low (10s timeout) 这些任务可以放后，但是最终应该得到执行。例如分析通知
- Idle (no timeout) 一些没有必要做的任务 (e.g. 比如隐藏的内容)。

目前 ContinuousEvent 对应的是 Immediate 优先级；UserBlockingEvent 对应的是 UserBlocking(需要手动开启)；而 DiscreteEvent 对应的也是 UserBlocking，只不过它在执行之前，先会执行完其他 Discrete 任务。

实现细节

现在开始进入文章正题，React 是怎么实现事件机制？主要分为两个部分：绑定和分发。

事件是如何绑定的？

为了避免后面绕晕了，有必要先了解一下 React 事件机制中的插件协议。每个插件的结构如下：

```
export type EventTypes = {[key: string]: DispatchConfig};
```

```
// 插件接口
```

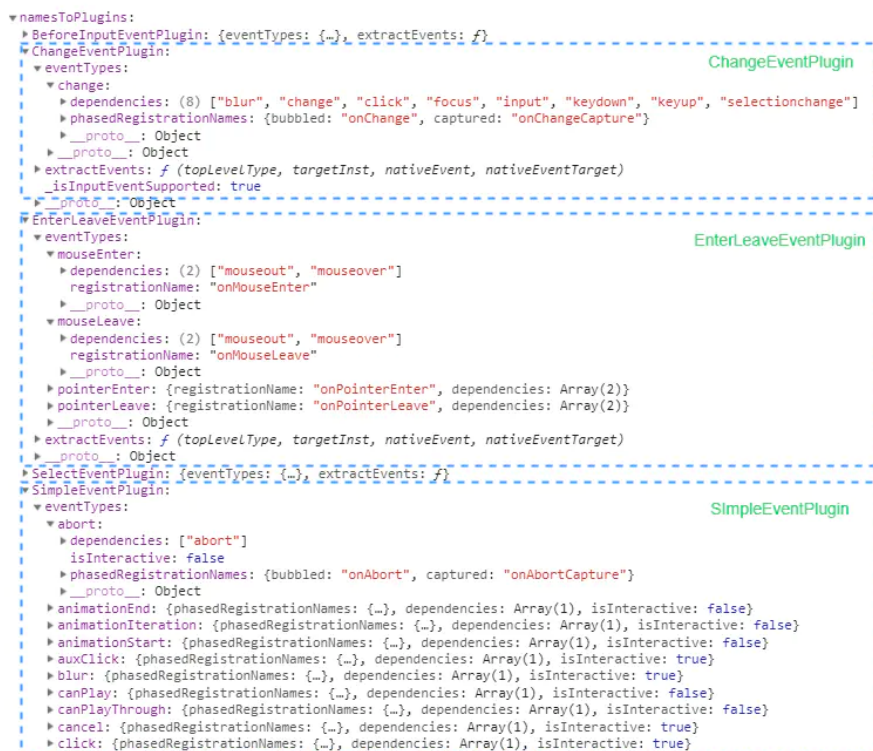
```
export type PluginModule<NativeEvent> = {
  eventTypes: EventTypes,           // 声明插件支持的事件类型
  extractEvents: (                  // 对事件进行处理，并返回合成事件
    对象
    topLevelType: TopLevelType,
    targetInst: null | Fiber,
    nativeEvent: NativeEvent,
    nativeEventTarget: EventTarget,
  ) => ?ReactSyntheticEvent,
  tapMoveThreshold?: number,
};
```

eventTypes 声明该插件负责的事件类型，它通过 DispatchConfig 来描述：

```
export type DispatchConfig = {
  dependencies: Array<TopLevelType>, // 依赖的原生事件，表示关联这些
  事件的触发。‘简单事件’一般只有一个，复杂事件如 onChange 会监听多个，
  如下图👉
```

```
    phasedRegistrationNames?: {      // 两阶段 props 事件注册名称，React
  会根据这些名称在组件实例中查找对应的 props 事件处理器
    bubbled: string,                 // 冒泡阶段，如 onClick
    captured: string,                // 捕获阶段，如 onClickCapture
  },
  registrationName?: string          // props 事件注册名称，比如
  onMouseEnter 这些不支持冒泡的事件类型，只会定义 registrationName，不
  会定义 phasedRegistrationNames
  eventPriority: EventPriority,      // 事件的优先级，上文已经介绍过了
};
```

看一下实例：



上面列举了三个典型的 EventPlugin:

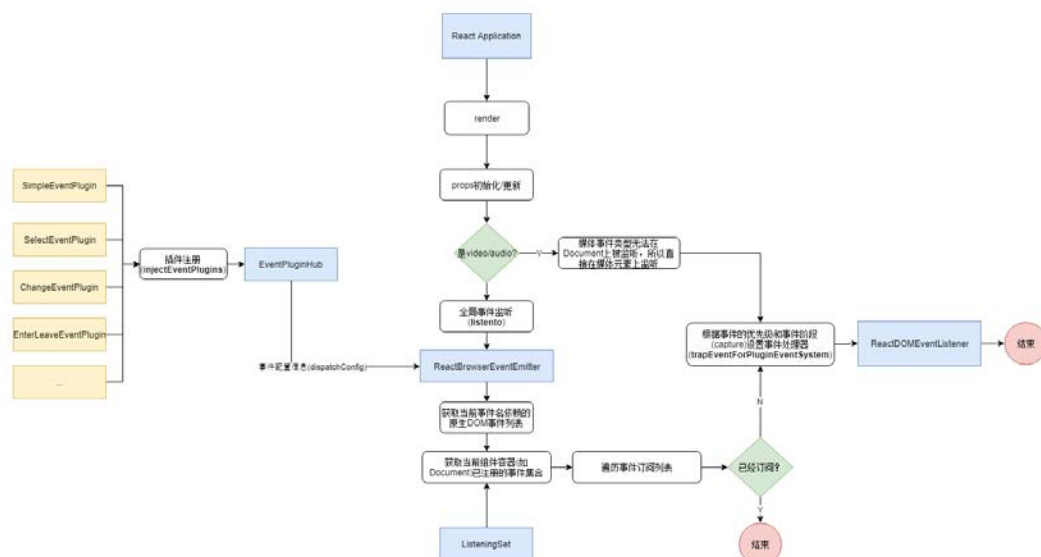
- **SimpleEventPlugin** - 简单事件最好理解，它们的行为都比较通用，没有什么 Trick，例如不支持事件冒泡、不支持在 Document 上绑定等等。和原生 DOM 事件是一一对应的关系，比较好处理。
- **EnterLeaveEventPlugin** - 从上图可以看出来，mouseenter 和 mouseLeave 依赖的是 mouseout 和 mouseover 事件。也就是说 *Enter/*Leave 事件在 React 中是通过*Over/*Out 事件来模拟的。这样做的好处是可以在 document 上面进行委托监听，还有避免 *Enter/*Leave 一些奇怪而不实用的行为。
- **ChangeEventPlugin** - onChange 是 React 的一个自定义事件，可以看出它依赖了多种原生 DOM 事件类型来模拟 onChange 事件。

另外每个插件还会定义 extractEvents 方法，这个方法接受事件名称、原生 DOM 事件对象、事件触发的 DOM 元素以及 React 组件实例，返回一个合成事件对象，如果返回空则表示不作处理。关于 extractEvents 的细节会在下一节阐述。

在 ReactDOM 启动时就会向 EventPluginHub 注册这些插件：

```
EventPluginHubInjection.injectEventPluginsByName({
  SimpleEventPlugin: SimpleEventPlugin,
  EnterLeaveEventPlugin: EnterLeaveEventPlugin,
  ChangeEventPlugin: ChangeEventPlugin,
  SelectEventPlugin: SelectEventPlugin,
  BeforeInputEventPlugin: BeforeInputEventPlugin,
});
```

这里先看一下流程图，忽略杂乱的跳转：



- 1. 在 props 初始化和更新时会进行事件绑定。首先 React 会判断元素是否是媒体类型，媒体类型的事件是无法在 Document 监听的，所以会直接在元素上进行绑定
- 2. 反之就在 Document 上绑定。这里面需要两个信息，一个就是上文提到的‘事件依赖列表’，比如 onMouseEnter 依赖 mouseover/mouseout；第二个是 ReactBrowserEventEmitter 维护的‘已订阅事件表’。事件处理器只需在 Document 订阅一次，所以相比在每个元素上订阅事件会节省很多资源。

代码大概如下：

```
export function listenTo(
```

```

    registrationName: string,          // 注册名称, 如 onClick
    mountAt: Document | Element | Node, // 组件树容器, 一般是 Document
  ): void {
    const listeningSet = getListeningSetForElement(mountAt);
    // 已订阅事件表
    const dependencies =
      registrationNameDependencies[registrationName]; // 事件依赖

    for (let i = 0; i < dependencies.length; i++) {
      const dependency = dependencies[i];
      if (!listeningSet.has(dependency))
      {
        // 未订阅
        switch (dependency) {
          // ... 特殊的事件监听处理
          default:
            const isMediaEvent =
              mediaEventTypes.indexOf(dependency) !== -1;
            if (!isMediaEvent) {
              trapBubbledEvent(dependency, mountAt);
            }
            // 设置事件处理器
            break;
        }
        listeningSet.add(dependency);
      }
      // 更新已订阅表
    }
  }
}

```

- 接下来就是根据事件的‘优先级’和‘捕获阶段’ (是否是 capture) 来设置事件处理器:

```

function trapEventForPluginEventSystem(
  element: Document | Element | Node, // 绑定到元素, 一般是
  Document
  topLevelType: DOMTopLevelEventType, // 事件名称
  capture: boolean,
): void {
  let listener;
  switch (getEventPriority(topLevelType)) {
    // 不同优先级的事件类型, 有不同的事件处理器进行分发, 下文会详细介绍
    case DiscreteEvent: //  离散事件

```



```

    listener = dispatchDiscreteEvent.bind(
        null,
        topLevelType,
        PLUGIN_EVENT_SYSTEM,
    );
    break;

case UserBlockingEvent:                // ⚙️ 用户阻塞事件

    listener = dispatchUserBlockingUpdate.bind(
        null,
        topLevelType,
        PLUGIN_EVENT_SYSTEM,
    );
    break;

case ContinuousEvent:                  // ⚙️ 可连续事件

default:
    listener = dispatchEvent.bind(null, topLevelType,
    PLUGIN_EVENT_SYSTEM);
    break;
}

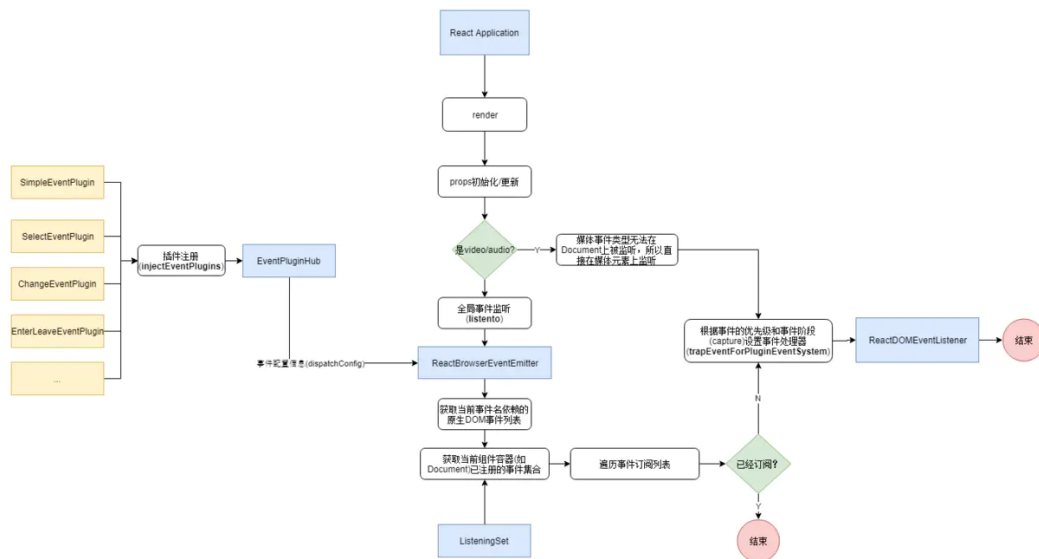
const rawEventName = getRawEventName(topLevelType);
if (capture) {                          // 绑定事件处理器到元素
    addEventCaptureListener(element, rawEventName, listener);
} else {
    addEventBubbleListener(element, rawEventName, listener);
}
}

```

事件绑定的过程还比较简单，接下来看看事件是如何分发的。

事件是如何分发的？

按惯例还是先上流程图：



事件触发调度

通过上面的 `trapEventForPluginEventSystem` 函数可以知道，不同的事件类型有不同的事件处理器，它们的区别是调度的优先级不一样：

// 离散事件

// `discrentUpdates` 在 `UserBlocking` 优先级中执行

```
function dispatchDiscreteEvent(topLevelType, eventSystemFlags,
nativeEvent) {
  flushDiscreteUpdatesIfNeeded(nativeEvent.timeStamp);
  discreteUpdates(dispatchEvent, topLevelType, eventSystemFlags,
nativeEvent);
}
```

// 阻塞事件

function dispatchUserBlockingUpdate(

```
  topLevelType,
  eventSystemFlags,
  nativeEvent,
```

) {

// 如果开启了 `enableUserBlockingEvents`，则在 `UserBlocking` 优先级中调度，

// 开启 `enableUserBlockingEvents` 可以防止饥饿问题，因为阻塞事件中有 `scroll`、`mousemove` 这类频繁触发的事件

// 否则同步执行

```
if (enableUserBlockingEvents) {
```

```

    runWithPriority(
      UserBlockingPriority,
      dispatchEvent.bind(null, topLevelType, eventSystemFlags,
nativeEvent),
    );
  } else {
    dispatchEvent(topLevelType, eventSystemFlags, nativeEvent);
  }
}

// 可连续事件则直接同步调用 dispatchEvent

```

最终不同的事件类型都会调用 `dispatchEvent` 函数。 `dispatchEvent` 中会从 DOM 原生事件对象获取事件触发的 `target`，再根据这个 `target` 获取关联的 React 节点实例。

```

export function dispatchEvent(topLevelType: DOMTopLevelEventType,
eventSystemFlags: EventSystemFlags, nativeEvent: AnyNativeEvent):
void {
  // 获取事件触发的目标 DOM
  const nativeEventTarget = getEventTarget(nativeEvent);
  // 获取离该 DOM 最近的组件实例(只能是 DOM 元素组件)
  let targetInst = getClosestInstanceFromNode(nativeEventTarget);
  // ....
  dispatchEventForPluginEventSystem(topLevelType, eventSystemFlags,
nativeEvent, targetInst);
}

```

接着(中间还有一些步骤，这里忽略)会调用 `EventPluginHub` 的 `runExtractedPluginEventsInBatch`，这个方法遍历插件列表来处理事件，生成一个 `SyntheticEvent` 列表：

```

export function runExtractedPluginEventsInBatch(
  topLevelType: TopLevelType,
  targetInst: null | Fiber,
  nativeEvent: AnyNativeEvent,
  nativeEventTarget: EventTarget,
) {
  // 遍历插件列表，调用插件的 extractEvents，生成 SyntheticEvent 列表
  const events = extractPluginEvents(
    topLevelType,

```

```

    targetInst,
    nativeEvent,
    nativeEventTarget,
  );

  // 事件处理器执行，见后文批量执行
  runEventsInBatch(events);
}

```

插件是如何处理事件？

现在来看看插件是如何处理事件的，我们以 SimpleEventPlugin 为例：

```

const SimpleEventPlugin: PluginModule<MouseEvent> & {
  getEventPriority: (topLevelType: TopLevelType) => EventPriority,
} = {
  eventTypes: eventTypes,
  // 抽取事件对象
  extractEvents: function(
    topLevelType: TopLevelType,
    targetInst: null | Fiber,
    nativeEvent: MouseEvent,
    nativeEventTarget: EventTarget,
  ): null | ReactSyntheticEvent {
    // 事件配置
    const dispatchConfig =
topLevelEventsToDispatchConfig[topLevelType];

    //❶ 根据事件类型获取 SyntheticEvent 子类事件构造器
    let EventConstructor;
    switch (topLevelType) {
      // ...
      case DOMTopLevelEventTypes.TOP_KEY_DOWN:
      case DOMTopLevelEventTypes.TOP_KEY_UP:
        EventConstructor = SyntheticKeyboardEvent;
        break;
      case DOMTopLevelEventTypes.TOP_BLUR:
      case DOMTopLevelEventTypes.TOP_FOCUS:
        EventConstructor = SyntheticFocusEvent;
        break;
      // ... 省略
      case DOMTopLevelEventTypes.TOP_GOT_POINTER_CAPTURE:

```

```

    // ...
    case DOMTopLevelEventTypes.TOP_POINTER_UP:
      EventConstructor = SyntheticPointerEvent;
      break;
    default:
      EventConstructor = SyntheticEvent;
      break;
  }

  //❷ 构造事件对象，从对象池中获取
  const event = EventConstructor.getPooled(
    dispatchConfig,
    targetInst,
    nativeEvent,
    nativeEventTarget,
  );

  //❸ 根据 DOM 事件传播的顺序获取用户事件处理器
  accumulateTwoPhaseDispatches(event);
  return event;
},
};

```

SimpleEventPlugin 的 extractEvents 主要做以下三个事情：

- ❶ 根据事件的类型确定 SyntheticEvent 构造器
- ❷ 构造 SyntheticEvent 对象。
- ❸ 根据 DOM 事件传播的顺序获取用户事件处理器列表

为了避免频繁创建和释放事件对象导致性能损耗(对象创建和垃圾回收)，React 使用一个事件池来负责管理事件对象，使用完的事件对象会放回池中，以备后续的复用。

这也意味着，在事件处理器同步执行完后，SyntheticEvent 对象就会马上被回收，所有属性都会无效。所以一般不会在异步操作中访问 SyntheticEvent 事件对象。你也可以通过以下方法来保持事件对象的引用：

- 调用 SyntheticEvent#persist() 方法，告诉 React 不要回收对象池
- 直接引用 SyntheticEvent#nativeEvent，nativeEvent 是可以持久引用的，不过为了不打破抽象，建议不要直接引用 nativeEvent

构建完 SyntheticEvent 对象后，就需要遍历组件树来获取订阅该事件的用户事件处理器了：

```
function accumulateTwoPhaseDispatchesSingle(event) {  
  // 以_targetInst 为基点，按照 DOM 事件传播的顺序遍历组件树  
  traverseTwoPhase(event._targetInst,  
    accumulateDirectionalDispatches, event);  
}
```

遍历方法其实很简单：

```
export function traverseTwoPhase(inst, fn, arg) {  
  const path = [];  
  while (inst) { // 从 inst 开始，向上级回溯  
    path.push(inst);  
    inst = getParent(inst);  
  }  
  
  let i;  
  // 捕获阶段，先从最顶层的父组件开始，向下级传播  
  for (i = path.length; i-- > 0; ) {  
    fn(path[i], 'captured', arg);  
  }  
  
  // 冒泡阶段，从 inst，即事件触发点开始，向上级传播  
  for (i = 0; i < path.length; i++) {  
    fn(path[i], 'bubbled', arg);  
  }  
}
```

accumulateDirectionalDispatches 函数则是简单查找当前节点是否有对应的事件处理器：

```
function accumulateDirectionalDispatches(inst, phase, event) {  
  // 检查是否存在事件处理器  
  const listener = listenerAtPhase(inst, event, phase);  
  // 所有处理器都放入到_dispatchListeners 队列中，后续批量执行这个队列  
  if (listener) {  
    event._dispatchListeners = accumulateInto(  
      event._dispatchListeners,  
      listener,  
    );  
  }  
}
```

```

    event._dispatchInstances =
    accumulateInto(event._dispatchInstances, inst);
  }
}

```

例如下面的组件树，遍历过程是这样的：



最终计算出来的`_dispatchListeners` 队列是这样的：`[handleB, handleC, handleA]`

批量执行

遍历执行插件后，会得到一个 `SyntheticEvent` 列表，`runEventsInBatch` 就是批量执行这些事件中的`_dispatchListeners` 事件队列

```

export function runEventsInBatch(
  events: Array<ReactSyntheticEvent> | ReactSyntheticEvent | null,
) {
  // ...
  forEachAccumulated(processingEventQueue,
    executeDispatchesAndRelease);
}

```

```
// 📌
```

```

const executeDispatchesAndRelease = function(event:
ReactSyntheticEvent) {

```

```

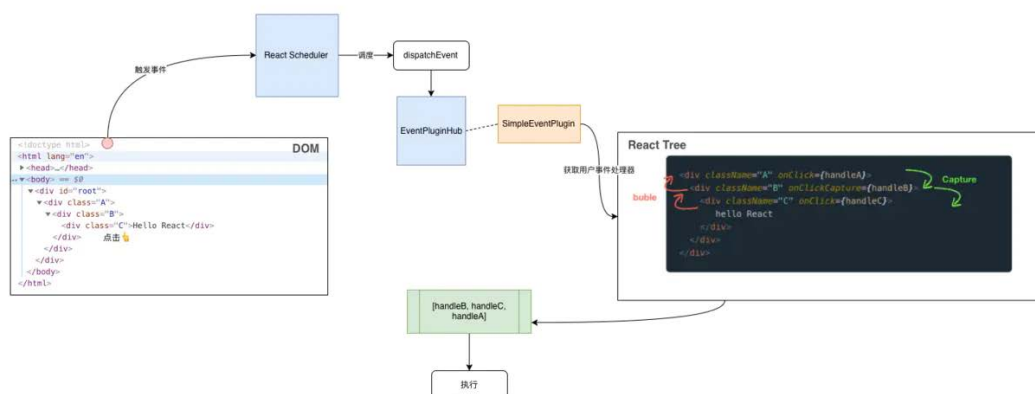
if (event) {
  // 按顺序执行_dispatchListeners

  // 📌
  executeDispatchesInOrder(event);

  // 如果没有调用 persist() 方法则直接回收
  if (!event.isPersistent()) {
    event.constructor.release(event);
  }
}
};

export function executeDispatchesInOrder(event) {
  // 遍历 dispatchListeners
  for (let i = 0; i < dispatchListeners.length; i++) {
    // 通过调用 stopPropagation 方法可以禁止执行下一个事件处理器
    if (event.isPropagationStopped()) {
      break;
    }
    // 执行事件处理器
    executeDispatch(event, dispatchListeners[i],
dispatchInstances[i]);
  }
}

```



OK，到这里 React 的事件机制就基本介绍完了，这里只是简单了介绍了一下 SimpleEventPlugin，实际代码中还有很多事件处理的细节，限于篇幅，本文就不展开去讲了。有兴趣的读者可以亲自去观摩 React 的源代码。

未来

React 内部有一个实验性的事件 API，React 内部称为 React Flare、正式名称是 [react-events](#)，通过这个 API 可以实现跨平台、跨设备的高级事件封装。

react-events 定义了一个**事件响应器 (Event Responders)**的概念，这个事件响应器可以捕获子组件树或应用根节点的事件，然后转换为自定义事件。

比较典型的高级事件是 press、longPress、swipe 这些手势。通常我们需要自己或者利用第三方库来实现这一套手势识别，例如

```
import Gesture from 'rc-gesture';
```

```
ReactDOM.render(  
  <Gesture  
    onTap={handleTap}  
    onSwipe={onSwipe}  
    onPinch={handlePinch}  
  >  
    <div>container</div>  
  </Gesture>,  
  container);
```

那么 react-events 的目的就是**提供一套通用的事件机制给开发者来实现‘高级事件’的封装，甚至实现事件的跨平台、跨设备**，现在你可以通过 react-events 来封装这些手势事件。

react-events 除了核心的 Responder 接口，还封装了一些内置模块，实现跨平台的、常用的高级事件封装：

- Focus module
- Hover module
- Press module
- FocusScope module
- Input module
- KeyBoard module
- Drag module
- Pan module


- Scroll module
- Swipe module

举 Press 模块作为例子, [Press 模块](#) 会响应它包裹的元素的 press 事件。press 事件包括 onContextMenu、onLongPress、onPress、onPressEnd、onPressMove、onPressStart 等等。其底层通过 mouse、pen、touch、trackpad 等事件来转换。

看看使用示例:

```
import { PressResponder, usePressListener } from 'react-
events/press';

const Button = (props) => (

  const listener = usePressListener({ //  通过 hooks 创建 Responder
    onPressStart,
    onPress,
    onPressEnd,
  })

  return (
    <div listeners={listener}>
      {subtrees}
    </div>
  );
);
```

react-events 的运作流程图如下, 事件响应器 (Event Responders) 会挂载到 host 节点, 它会在 host 节点监听 host 或子节点分发的原生事件 (DOM 或 React Native), 并将它们转换/合并成高级的事件:

你可以通过这个 Codesandbox 玩一下 react-events:

初探 Responder 的创建

我们挑一个简单的模块来了解一些 react-events 的核心 API，目前最简单的是 Keyboard 模块。Keyboard 模块的目的就是规范化 keydown 和 keyup 事件对象的 key 属性(部分浏览器 key 属性的行为不一样)，它的实现如下：

```
/**
 * 定义 Responder 的实现
 */
const keyboardResponderImpl = {
  /**
   * ①定义 Responder 需要监听的子树的 DOM 事件，对于 Keyboard 来说是
   ['keydown', 'keyup'];
   */
  targetEventTypes,
  /**
   * ②监听子树触发的事件
   */
  onEvent(
    event: ReactDOMResponderEvent, // 包含了当前触发事件的相关信息，如原生事件对象，事件触发的节点，事件类型等等
    context: ReactDOMResponderContext, // Responder 的上下文，给 Responder 提供了一些方法来驱动事件分发
    props: KeyboardResponderProps, // 传递给 Responder 的 props
  ): void {
    const {responderTarget, type} = event;

    if (props.disabled) {
      return;
    }

    if (type === 'keydown') {
      dispatchKeyboardEvent(
        'onKeyDown',
        event,
        context,
        'keydown',
        ((responderTarget: any): Element | Document),
      );
    } else if (type === 'keyup') {
      dispatchKeyboardEvent(
        'onKeyUp',
        event,
```

```

        context,
        'keyup',
        ((responderTarget: any): Element | Document),
    );
}
},
};

```

再看看 dispatchKeyboardEvent:

```

function dispatchKeyboardEvent(
    eventPropName: string,
    event: ReactDOMResponderEvent,
    context: ReactDOMResponderContext,
    type: KeyboardEventType,
    target: Element | Document,
): void {
    // 🌀 创建合成事件对象，在这个函数中会规范化事件的 key 属性
    const syntheticEvent = createKeyboardEvent(event, context, type,
    target);

    // 🌀 通过 Responder 上下文分发事件
    context.dispatchEvent(eventPropName, syntheticEvent,
    DiscreteEvent);
}

```

导出 Responder:

```

// 🌀 createResponder 把 keyboardResponderImpl 转换为组件形式
export const KeyboardResponder = React.unstable_createResponder(
    'Keyboard',
    keyboardResponderImpl,
);

// 🌀 创建 hooks 形式
export function useKeyboardListener(props: KeyboardListenerProps):
void {
    React.unstable_useListener(KeyboardResponder, props);
}

```

现在读者应该对 **Responder** 的职责有了一些基本的了解，它主要做以下几件事情：

- 声明要监听的原生事件(如 DOM)，如上面的 `targetEventTypes`
- 处理和转换合成事件，如上面的 `onEvent`
- 创建并分发自定义事件。如上面的 `context.dispatchEvent`

和上面的 Keyboard 模块相比，现实中的很多高级事件，如 `longPress`，它们的实现则要复杂得多。它们可能要维持一定的**状态**、也可能要独占响应的**所有权**（即同一时间只能有一个 Responder 可以对事件进行处理，这个常用于移动端触摸手势

`react-events` 目前都考虑了这些场景，看一下 API 概览：

react-events 意义何在？

上文提到了 React 事件内部采用了插件机制，来实现事件处理和合成，比较典型的就是 `onChange` 事件。`onChange` 事件其实就是所谓的‘高级事件’，它是通过表单组件的各种原生事件来模拟的。

也就是说，React 通过插件机制本质上是可以实现高级事件的封装的。但是如果读者看过源代码，就会觉得里面逻辑比较绕，而且依赖 React 的很多内部实现。**所以这种内部的插件机制并不是面向普通开发者的。**

`react-events` 接口就简单很多了，它屏蔽了很多内部细节，面向普通开发者。我们可以利用它来实现高性能的自定义事件分发，更大的意义是通过它可以实现跨平台/设备的事件处理方式。

目前 `react-events` 还是实验阶段，特性是默认关闭，API 可能会出现变更，所以不建议在生产环境使用。可以通过这个来关注它的进展。

最后赞叹一下 React 团队创新能力！

完！